

Computer Science Department

TECHNICAL REPORT

Efficient Robust Parallel Computations

Zvi M. Kedem
Krishna V. Palem
Paul Spirakis

Technical Report 472

November 1989

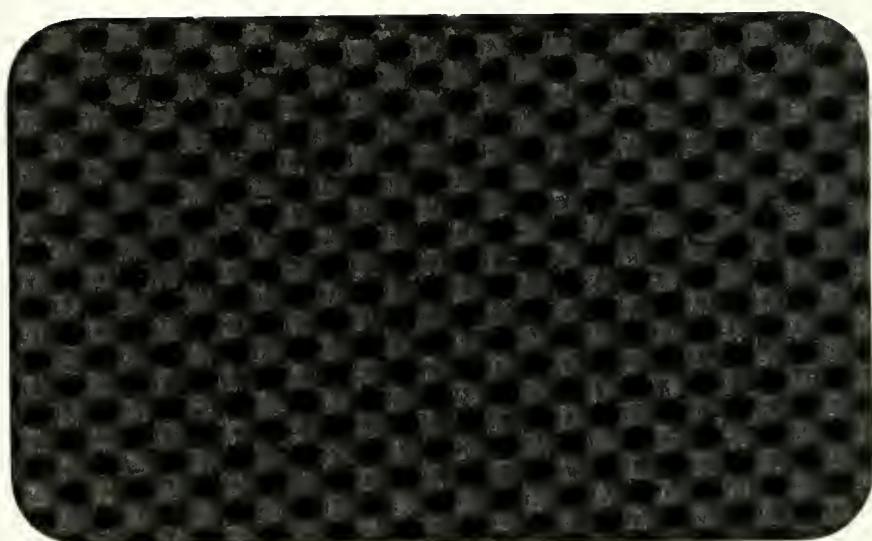
NEW YORK UNIVERSITY
COURANT INSTITUTE LIBRARY
251 MERCER ST. NEW YORK, N.Y. 10012

NEW YORK UNIVERSITY



Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

NYU COMPSCI TR-472
Kedem, Zvi M
Efficient robust parallel
computations.
c.1



Efficient Robust Parallel Computations

Zvi M. Kedem
Krishna V. Palem
Paul Spirakis

Technical Report 472

November 1989

RECEIVED
COURANT INSTITUTE LIBRARY
251 Mercer St. New York, N.Y. 10012

Efficient Robust Parallel Computations

(Extended Abstract)

*Zvi M. Kedem **

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
New York, NY 10012-1185

Krishna V. Palem

IBM Research Division
T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Paul Spirakis

Computer Technology Institute
Patras University
Patras
and Courant Institute

November 16, 1989

Abstract

A parallel computing system becomes increasingly prone to failure as the number of processing elements in it increases. In this paper, we describe a completely general strategy that takes an *arbitrary* step of an ideal CRCW PRAM and *automatically* translates it to run efficiently and robustly on a PRAM in which processors are prone to failure. The strategy relies on efficient robust algorithms for solving a core problem, the *Write-All Problem*. This problem is critical, because, as we show, its complexity is equal to that of any general strategy for realizing robustness in our model. We analyze the expected parallel time and work of various algorithms for solving this problem and prove a lower bound. Our general strategy is also applicable in the context of asynchronous PRAMs given asynchronous algorithms for the Write-All problems.

*The research of this author was supported in part by the Office of Naval Research under contract number N00014-85-K-0046 and by the National Science Foundation under grant number CCR-89-6949.

1 Introduction

With hardware becoming cheaper, it is expected that “massively parallel systems” with large numbers of processors will both increase the speed of computations and decrease their cost. Unfortunately, as the number of processing elements grows, certain difficulties need to be addressed, among them *asynchrony* and *processor faults*. These two problems are related, but in this paper we concentrate on processor faults only. Clearly, the larger the number of processors, the greater the probability of some processors failing.

Much of the recent emphasis in algorithm design for PRAMs for a variety of basic problems such as list ranking [TV84,CV86], sorting [C86], forest matching [KP88], pattern matching [KLP89] and others, has emphasized efficiency and optimal speedup. These extremely efficient (ideally optimal speedup) parallel algorithms have very little “slack” in that every step of the algorithm is essential. Therefore, quite often, they do not terminate correctly when perturbed by a few simple processor failures.

Because of this, it is of paramount importance to study the design of *robust and efficient* parallel algorithms. In this paper it is our goal to develop a general methodology for *implementing ideal CRCW PRAMs robustly on faulty CRCW PRAMs*. We adopt the approach of “graceful degradation,” so that if the processors fail-stop during the computation, as long as at least one processor remains operational, the PRAM program can continue executing correctly *independent of its semantics*. In addition, we want to minimize the performance penalty incurred in robust implementations. We present a technique that takes an *arbitrary* step of an ideal CRCW PRAM designed to run on U “virtual” processors, and execute it on a CRCW PRAM of the same type with $P \leq U$ processors that are prone to failure¹. (The number of virtual processors U , that is the parallelism width, can vary from step to step.) By modeling these failures probabilistically, we analyze the expected *work* and *parallel time* complexities of a variety of deterministic and probabilistic schemes for realizing these robust implementations.

Kanellakis and Shvartsman [KS89] were the first to formalize this notion of robustness within the context of synchronous parallel computation. They developed a failure model (to be described later), which we use here. In [KS89], they considered the *Write-All* problem, defined below, and described its deterministic robust implementation.

Write-All problem: *Given an array $x[1..U]$ initialized to 0, set $x[i] := 1$ for all i . (We will also refer to such writing of 1 as “marking.”)*

Their algorithm (henceforth referred to as the *KS-algorithm* for short) iteratively estimates the amount of remaining unwritten locations and remaining processors, and reschedules the processors to other locations for writing 1’s. It is easy to see that their algorithm for the Write-All problem can be used for computing associative functions, such as *max*.

Using the Write-All algorithm, Kanellakis and Shvartsman, designed robust algorithms for fundamental problems such as list ranking. In [KS89] they also analyze the deterministic worst case complexity of the KS-algorithm. They also observed various specific complexity improvements in the case where $P < U$. Specifically, they parameterized the KS-algorithm to achieve deterministic optimal work of $O(U)$ using $O(U/\log^2 U)$ initial processors. Improvement for the case where the number of initial processors is $U/\log U$ was observed independently by

¹That is we execute an ideal Common/Arbitrary/Priority CRCW PRAM program respectively on a Common/Arbitrary/Priority faulty CRCW PRAM.

Khuller [Kh89].

Martel et al. [MPS89] describe a probabilistic algorithm for computing the maximum of U elements assuming asynchronous computations. Their algorithm (to which we refer to as a *Collective Coupon Collector Algorithm* or *CCC-algorithm* for short) can be immediately used for robust computation of the maximum as well as for solving the Write-All problem. A variety of an asynchronous parallel model (APRAM) was described by Cole and Zajicek [CZ89], who presented algorithms for summation, graph connectivity, etc. on this model. Many researchers have studied closely related issues, including robustness and fault tolerance in an asynchronous setting [A88,AAG87,AS88,DPPU86,P85,SS83].

Previous efforts at “adapting” ideal PRAM programs to cope with imperfections, such as asynchrony or processor faults, have focussed on redesigning specific algorithms. Our paper departs significantly from this approach, by providing a general strategy for simulating arbitrary PRAM steps on PRAMs with faults. *Specifically, we show that the complexities of solving the Write-All problem robustly, and implementing a step of an ideal CRCW PRAM robustly are identical up to small constant multiplicative time and small constant per processor additive space overheads.* We do this constructively, by showing how to use an arbitrary robust Write-All algorithm twice² to implement an arbitrary step of the ideal PRAM. In effect we provide a *two-phase idempotent execution strategy* (*TIES* for short) that uses *any* robust algorithm for solving the Write-All problem³, to automatically yield robust implementations of arbitrary ideal PRAM steps. Independently [Shv89], Shvartsman obtained a technique, similar to our strategy described later, which allowed him to implement robustly any parallel algorithm whose work is within a polylog factor of that of the best sequential algorithm and whose local memory requirements are within a polylog factor of the problem size.

We identify the Write-All problem as the core step in robust implementations of PRAM algorithms. Therefore, solving it efficiently is critical to realizing such robust implementations with low overhead. Towards this end, we also introduce a new and extremely simple algorithm based on pointer-doubling (the *PD-algorithm*) for solving the Write-All problem. We analyze the expected work as well as the expected parallel time taken by this simple PD-algorithm algorithm, as well by the KS-algorithm. We show that expected behavior of the PD-algorithm is better than that of the KS-algorithm; detailed statements of these results can be found later. We also show that the expected work done by a *randomized* parallel version of a Coupon Collector scheme is less than that of the deterministic KS and PD algorithms. While the algorithms we study are not always complicated, we found the complexity analysis of their expected behavior to be technically intricate. Finally, we improve the lower bound (from [KS89]) on the amount of work that any *deterministic* algorithm for the Write-All algorithm must do in the worst case; this lower bound matches the expected work done by our deterministic PD-algorithm.

²Two points are in order here. First, we assume that termination of the Write-All algorithm is detectable. Second, we actually use a modification of the underlying Write-All algorithm as a “skeleton” for our technique. These notions will be made precise later.

³Actually, *any* algorithm that can verify that all the the given U instructions have been touched or executed, will suffice. The Write-All problem is a special case of this more general *Touch-All paradigm*, and is used here. Computation of maximum or summation could have been used also.

2 Model of failures and some conventions

Although our results hold for arbitrary CRCW PRAMs, for the purpose of this paper, we assume the Common variant of the CRCW PRAM of [FW78]. When dealing with probabilistic algorithms, we assume that the individual processors of the PRAM are supplied with independent random number generators.

Consider a parallel algorithm A that starts with $P_0 = P$ available processors and in parallel time τ completes its computation on input data I_A . The *availability pattern* Π_A indicates which processors are available or operational for the algorithm A at each parallel time step. Let P_i be the number of processors available at time i ($0 \leq i \leq \tau$). Then,

Definition 1 *The work performed in the execution of algorithm A on input I_A , with availability pattern Π is $W(A, I_A, \Pi) = \sum_{i=0}^{\tau} P_i$.⁴*

As in the case of Kanellakis and Shvartsman [KS89], we will only be interested in the *fail-stop* availability patterns. In such patterns, processors either stay available or *fail* and once they fail, they stay failed for the remaining steps of the computation.

An availability pattern Π is called *oblivious* if it is determined before the start of the execution of the algorithm. It is called *Byzantine* if it is created (by an adversary), adaptively, during the computation. In such a case, the adversary determines the set of available processors at the next time step by examining the history of the computation (which includes random choices previously done) up to that time.

The average case analysis in this paper deals with *random failures* where each processor may fail with a fixed probability $q < 1$ in a consecutive sequence of time steps, referred to here as an *epoch* or *period*; the size of each epoch may be a function of U . If all the processors in the PRAM fail, we assume that this condition can be detected externally.

3 Summary of Results

We now summarize our main contributions. All our expected case results (deterministic and randomized) summarized below hold with an extremely high probability of at least $1 - U^{-\gamma}$ for some $\gamma > 1$ controllable by the algorithm implementor.

3.1 The two-phase execution strategy

1. We introduce a two-phase idempotent execution strategy that, in conjunction with *any* robust algorithm for the Write-All problem, can be used to simulate an arbitrary step of a PRAM robustly. This TIES strategy can be implemented using a constant amount of additional space per functional processor.

⁴This notion of work generalizes the traditional processor-time product definition, for the case when there are no processor failures (see [TV84] for example). It is similar to the definition of work in [MPS89], although they define it in the slightly different but essentially equivalent model of asynchronous PRAMs. It is also the same as the *available processor steps* of [KS89].

3.2 Average case complexity results for the Write-All problem

We analyze the expected work as well as the expected parallel time of the KS-algorithm and the PD-algorithm for the Write-All problem. We assume a random pattern of failures as stated in section 2, where the failures can be Byzantine within each epoch.

2. Average case analysis of the KS-algorithm for the Write-All problem, with an epoch size of $\Theta(\log U)$. We show that:
 - (a) Its expected work is $O((P + U) \log U)$, where U is the array size and P is the initial number of processors.
 - (b) Its expected parallel time when $U = P$, and the processor failure rate is small, is $O(\log P \log U)$.
 - (c) Its expected parallel time when $P \leq c_0 U$ for some constant $c_0 > 0$ is $\Theta((U + \log P) \log U)$.

Note that the KS-algorithm has a very interesting threshold in its expected parallel time, which degrades very rapidly from $O(\log P \log U)$ (case (b) above) to *essentially sequential behavior* (case (c)) even when the number of initial processors P is $o(U)$.

3. We present a new and extremely simple PD-algorithm for the Write-All problem which is based on pointer doubling⁵. With an epoch size of $\log(U)$ we show that:
 - (a) The expected parallel time of the PD-algorithm is $O((U/P) \log U)$ when $U/\log U \leq P \leq U$.
 - (b) Hence its expected work is $O(U \log U)$.

Surprisingly, this simple PD-algorithm does significantly better in its expected parallel time behavior than the KS-algorithm. In particular, its expected parallel time stays stable at $O((U/P) \log U)$ even when the initial number of processors decreases to $P \geq U/\log U$.

3.3 Randomized algorithms for the Write-All problem

In this analysis, we assume worst-case oblivious availability patterns of processors.

4. We present a simple probabilistic algorithm for the Write-All problem, referred to here as the *Concurrent Coupon Collector Algorithm* (or *CCC-algorithm for short*, sketched in section 7.1) based on an approach of Martel et al. [MPS89]. We show a threshold behavior in the expected parallel time of this probabilistic algorithm as well. (Unfortunately, as noted in [MPS89], its expected work is $O(U \log U)$ and therefore it does not improve on the expected work of the KS or the PD-algorithms.)

⁵Initially, every location $i < U$ in x has a pointer to location $i + 1$. As soon as a 1 is written in location i , its pointer is used by the processor to find the next location to write. On subsequent steps, processors use pointers to double recursively. (A small initial segment of the array, is handled separately later.)

5. We show that by a minor modification to the CCC-algorithm, the resulting *Adapted Coupon Collector Algorithm* (or *ACC-algorithm* sketched in section 7.2) does only $O(U \log \log U)$ work provided $P \leq U/\log U$. Therefore, its expected work improves on that of the two deterministic algorithms described above even with as many as $U/\log U$ initial processors. Martel et al. [MPS89] analyze the expected work complexity of this algorithm as well, but their result appears to be flawed.

3.4 Lower bounds on worst-case work

6. We show that the work done by any algorithm for solving the Write-All problem is $\Omega(U + P \log U)$.

4 Two-phase idempotent execution strategy

4.1 Model of the ideal PRAM

Without loss of generality, our ideal CRCW PRAM is a simple modification of the PRAM as described [AHU74]. To save space, we omit many details. The PRAM has U “virtual” processors: $p_1, \dots, p_U$ with *no local* registers; all memory is *global* and *shared*. The “application memory” accessible by the processors is a shared vector $M[1..]$ of some length. W.l.o.g., the processors execute a single program whose instructions are listed in a vector $INST[1..I]$. The instructions are as those in [AHU74], with appropriate modifications. Thus, a typical instruction might be: $M[j_1] := M[j_1] + M[j_2]$. There is also a vector $PC[1..U]$, initialized to 1, containing the program counters of the processors. In a single step of the PRAM, each processor p_i executes the following

Internal Program: Read $PC[i]$; if $PC[i] = 0$, then halt; else, read, “decode,” execute $INST[PC[i]]$, and write the new value of $PC[i]$.

4.2 The strategy

We assume the existence of some robust algorithm for the Write-All problem, which given a vector $x[1..U]$ initialized to 0 and an availability pattern Π , sets $x[i] := 1$ for $i = 1, \dots, U$ in τ steps. It uses some auxiliary data structure AUX of size $O(U)$, appropriately initialized, *all* of whose locations are written during the execution of the algorithm. Furthermore, at the end of its execution, ab it variable $DONE$ is set to TRUE. We implement a robust CRCW PRAM in a faulty CRCW PRAM of the same type. Here we assume the *Common* type PRAM. One step of the ideal PRAM will take τ steps⁶ with constant (small) space overhead per processor.

Our (faulty) CRCW PRAM, will contain M and $INST$; two versions of PC (of the original application program), AUX and $DONE$: $PC.old$, $PC.new$, $AUX.old$, $AUX.new$, $DONE.old$, and $DONE.new$; and two new vectors $LOC[1..U]$, $VAL[1..U]$. In addition, we will need space to store the instructions of the Write-All algorithm, x , its PC ’s (which have to be handled

⁶This is the same number of steps as that taken by the Write-All algorithm; however each step will be slightly more time consuming.

carefully), etc.; we do not elaborate on this (other subtleties are ignored too). Also, w.l.o.g. we assume that M is of the same length as before.)

We now sketch, very briefly, the simulation of one step of the ideal PRAM while omitting many important technical details. By informal induction we assume:

- The values of M , $INST$, $PC.old$ in the faulty PRAM are the same as those of M , $INST$, PC in the ideal PRAM.
- The value of $AUX.old$ at the beginning of the simulation is the same as that of AUX at the beginning of the execution of the robust algorithm for Write-All and $DONE.old$ is $FALSE$.

The robust PRAM proceeds in two phases, each based on a single execution of the robust Write-All algorithm. By the algorithm, the (live) processors are synchronized between the phases. The behavior of the PRAM is reminiscent of the deferred write approach in the *redo/no-undo* recovery protocol in database operating systems. Phase 1, in effect, creates the deferred writes, phase 2 installs them. It is important that the phases are idempotent, as processors may fail after doing some work, and the various structures may become inconsistent. We describe the two phases (not in complete detail) in turn.

4.2.1 Phase 1

Run the underlying robust Write-All algorithm. Just before the point where p_k is supposed to execute $x[i] := 1$, simulate the execution of $INST[PC.old[i]]$. However, if $INST[PC.old[i]]$ assigns v to $M[l]$, *instead* execute $LOC[i] := l$, $VAL[i] := v$; if the instruction does not write, set $LOC[i] := 0$. The new value of $PC[i]$ will be stored in $PC.new[i]$. *In order to avoid circular reduction*, we have to be concerned with “cleaning up” of AUX , which is easily done. *Before* $AUX.old$ ’s location is being modified, the corresponding location of $AUX.new$ is initialized. At the end, $DONE.new := FALSE$ and $DONE.old := TRUE$ are executed.

4.2.2 Phase 2

This phase starts when $DONE.old$ becomes $TRUE$. Again, using the robust Write-All algorithm, values computed in phase 1 are copied into correct locations: if $LOC[i] \neq 0$ then $M[LOC[i]] := VAL[i]$. Furthermore, $PC.old[i] := PC.new[i]$. $AUX.new$ is being used, and $AUX.old$ is being (re-)initialized. At the end, $DONE.old := FALSE$ and $DONE.new := TRUE$ are executed.

5 Average-case analysis of the KS-algorithm

Let $j_0, j_1, \dots$ be the time instants at which processors actually write into the array x ; these steps will be referred to as *writing epochs* or *epochs*. We use U_j to denote the number of unwritten locations at the beginning of step j . Our availability patterns Π are *random* in the sense that each processor, which is operational at the end of writing epoch j_x , may fail with fixed probability $q < 1$ between the epochs j_x and j_{x+1} . Π is byzantine otherwise. Let $P_0 = P$

be the number of processors initially available, and assume $U_0 = U$. We will first prove the following useful technical theorem.

Theorem 1 *There are constants $\epsilon_1, \epsilon_2 \in (0, 1)$, and $a > 1$ that depend on q , such that, as long as $P_{j_x} \geq a \log P$, the inequality $\epsilon_1 \leq P_{j_{x+1}}/P_{j_x} \leq \epsilon_2$ holds for all such epochs j_x with probability at least $1 - P^{-\gamma}$ for some $\gamma > 2$.*

Proof Given an epoch j_x such that the number of operational processors in the beginning of j_x is $P_{j_x} \geq a \log P$, and any $\beta \in (0, 1)$, we have from Chernoff bounds [C52] that if E_x is the event “the number of failed processors is in the interval $(1 \pm \beta)qP_{j_x}$,” then $\text{Prob}\{E_x\} \geq 1 - \exp(-\frac{\beta^2}{2}qP_{j_x})$. Choose any $a > \frac{6}{\beta^2 q}$ and let $\gamma = \frac{a\beta^2 q}{2} - 1$ ($\gamma > 2$). Then, $\text{Prob}\{E_x\} \geq 1 - P^{-(\gamma+1)}$.

Now, let E be the event that E_x holds for all epochs j_x such that $P_{j_x} \geq a \log P$. Then, if $\bar{E}$ is the event “there is a j_x such that E is violated,” we get $\text{Prob}\{\bar{E}\} \leq \sum \text{Prob}\{\bar{E}_x\}$ over all j_x for which $P_{j_x} \geq a \log P$, i.e. $\text{Prob}\{\bar{E}\} \leq \frac{U_0}{a \log P} P^{-(\gamma+1)} \leq P^{-\gamma}$. Hence, $\text{Prob}\{E\} \geq 1 - P^{-\gamma}$. This completes the proof of the Theorem with constants $\epsilon_1 = 1 - (1 + \beta)q$, $\epsilon_2 = 1 - (1 - \beta)q$. $\in$

For convenience of analysis, we will view the execution of algorithm KS as being split into two phases. The first phase contains all epochs j_x such that $P_{j_x} \geq a \log P$ and the second phase is the rest of the algorithm’s execution.

5.1 Analysis of the first phase

For the first phase we have:

Lemma 1 *Conditioned on event E , the number of epochs of the first phase is $O(\log P)$.*

Proof We have $P_{j_{x+1}} \leq \epsilon_2 P_{j_x}$ for all the epochs of the first phase. If the number of such epochs is e_1 , then it must be that $P_{j_{e_1+1}} < a \log P$, i.e. $P \epsilon_2^{(e_1+1)} \leq a \log P$, i.e. $e_1 = O(\log P)$. $\in$

Corollary 1 *Conditioned on event E , the amount of work for the first phase is $W_1 = O(P \log U)$.*

Corollary 2 *Conditioned on event E , the number of parallel steps of algorithm KS for the first phase is $T_1 = O(\log P \log U)$.*

5.2 Analysis of the second phase

To analyze the second phase of algorithm KS we need the following result of [KS89]:

Fact 1 *Let P_i and U_i respectively denote the number of operating processors and the number of unwritten positions at the beginning of epoch i . Then, the work from this epoch is $O((P_i + U_i + P_i \log U_i) \log U)$.*

By using this fact and Theorem 1, we get:

Lemma 2 *Conditioned on event E , the work in the second phase of algorithm KS is $W_2 = O((a \log P + U + a \log P \log U) \log U)$.*

Having estimated the work during the second phase, we now analyze its expected parallel time. To do this, we need to calculate the number of unwritten positions $U_{j_{\epsilon_1}}$ at the end of the first phase. Again, conditioning on the event E we get (by an easy calculation) that $U_{j_{\epsilon_1}} \geq U - \frac{P}{1-\epsilon_2}$. We distinguish two cases depending on the value of P :

5.2.1 Case 1: $U = P$ and $q < 1/3$.

Lemma 3 *Conditioned on event E , for any epoch j_x of the first phase it is true that $U_{j_x} \leq P_{j_x}$.*

Proof Conditioned on event E , we have $\frac{P_{j_1}}{P_{j_0}} \geq \epsilon_1$ (where $P_{j_0} = P_0$). Let $r_1 = \frac{P_{j_1}}{P_{j_0}}$. By an inequality of [KS89] we have $U_{j_1} \leq U_{j_0}(1 - \frac{r_1}{2})$. Thus, $U_{j_1} \leq U(1 - \frac{\epsilon_1}{2})$. Also, $P_{j_1} \geq \epsilon_1 P \geq \epsilon_1 U$. Thus, $U_{j_1} \leq P_{j_1}$ when $1 - \frac{\epsilon_1}{2} \geq \epsilon_1$, i.e. when $\epsilon_1 \geq 2/3$ i.e. when $q < 1/3$. By repeating the above argument (for $q < 1/3$) the Lemma is proved. $\in$

By Lemma 3 and given the event E , we get $U_{j_{\epsilon_1}} = O(\log P)$, for which we have:

Lemma 4 *If $U \leq P$ and $q < 1/3$, then with probability at least $1 - P^{-\gamma}$ for some $\gamma > 2$, the number of parallel steps of the second phase of algorithm KS is $T_2 = O(\log P \log U)$ and hence its parallel time is $O(\log P \log U)$ with the same probability.*

5.2.2 Case 2: $\frac{P}{1-\epsilon_2} \leq \lambda U$, $0 < \lambda < 1$.

Then, given that E holds, we get $U_{j_{\epsilon_1}} \geq (1 - \lambda)U$ and $P_{j_{\epsilon_1}+1} < a \log P$. In this case, the number of epochs of the second phase is $O(U)$ since the processors are very few and the number of unwritten positions is very large. Thus

Lemma 5 *If there is a constant $\lambda \in (0, 1)$ such that $P \leq \lambda(1 - \epsilon_2)U$ (ϵ_2 as in Theorem 1) then, with probability at least $1 - P^{-\gamma}$ for some $\gamma > 2$, the number of parallel steps of the second phase of the KS-algorithm is $T_2 = \Theta(U \log U)$ and hence its parallel time is $\Theta(U \log U)$ with the same probability.*

5.3 The expected work and parallel time of the KS-algorithm

From Corollary 2 and Lemma 2, the claimed result (2(a) from section 3) about the expected work of the KS-algorithm follows. As for the claimed results 2(b) and 2(c) (also from section 3) about the expected parallel time of algorithm-KS, Corollary 1 together with Lemmas 3 and 4 yield result 2(b), whereas Lemma 5 yields 2(c).

6 The PD-algorithm and its average case analysis

6.1 The algorithm

We propose the following simple algorithm that exhibits stable expected parallel time behavior, and has low expected work complexity. The PD-algorithm involves a trivial modification of

the (well-known) naive pointer-doubling technique. Basically, in addition to the shared array $x[1..U]$ of the Write-All problem, we assume that each array location also contains a pointer. The pointers can be implemented as an additional array $s[1..U]$. Initially, $s[i] = i + 1$ for $i < U$ and $s[U] = \text{nil}$. Furthermore, $DONE$ is initialized to $FALSE$.

Algorithm Pointer-Doubling

- *Processor assignment:* For each k ($1 \leq k \leq P$) processor k is assigned to write 1 in array position $x[i]$ where $i = (k - 1)\frac{U}{P} + 1$.
- *Phase 1:* Each processor k executes the following loop:
`while  $s[i] \neq \text{nil}$  do`
`if  $x[s[i]] = 0$  then  $x[s[i]] := 1$ ;  $s[i] := s[s[i]]$ ;`
`od`
- *Phase 2:* if $s[i] = \text{nil}$ then
go to the head of the list and execute phase 1 until nil is encountered, execute $DONE := TRUE$, and halt.

Lemma 6 (Correctness) *If at least one processor survives, all of $x[1..U]$ will be initialized to 1.*

6.2 Average case analysis of the PD-algorithm

In the sequel we will assume that the initial number of processors is $P \geq U/\log U$. We will assume an availability pattern Π which is *random*. As before, we split the parallel time into consecutive intervals or *periods* where each period has $\log U$ steps. Each processor which is operational at the beginning of a period has a fixed probability $q < 1$ of failing at some step during that interval, independently of other processors. Notice that the availability pattern assumed here is equivalent to that assumed in the average case analysis of the KS-algorithm in the sense that the processor “decay rate” is constant, measured over periods of $\log U$ steps. Patterns with more frequent failure (faster decay rates) than Π are unacceptable because they tend to exhaust the available processors before any useful work can be done. We analyze the complexity of the PD-algorithm by considering the following two cases.

6.2.1 Case 1: $P = U$.

In this case, each array location is assigned to a processor initially. We first show that

Lemma 7 *There exists a fixed $\alpha > 1$ such that after α periods, except for an initial segment of length $O(\log U)$, the array x will be marked with probability at least $1 - U^{-b}$, where $b > 1$ depends on α .*

Let us now condition the rest of the analysis on the event described in Lemma 7. It follows that the PD-algorithm will terminate in $O(\log U)$ steps with probability $1 - U^{-b}$ for some $b > 1$. This is because in an additional amount of $O(\log U)$ steps, the number of remaining processors will still be a constant fraction of U , and at least one of them will mark the (possibly) unmarked initial segment of the array x . Thus,

Theorem 2 *The expected parallel time of the PD-algorithm, with U processors initially is $O(\log U)$. The probability that the actual parallel time exceeds $\Theta(\log U)$ is less than U^{-b} , for some $b > 1$.*

6.2.2 Case 2: $\lambda \frac{U}{\log U} \leq P < U$ for some constant $\lambda > 1$.

Recall that initially, each processor k ($1 \leq k \leq P$) is assigned to array location $(k-1)\frac{U}{P} + 1$. As in case 1 above (Lemma 7), we have,

Lemma 8 *Given $b > 1$ we can select an $\alpha > 1$ so that, after α periods and with probability at least $1 - U^{-b}$, the number of operational processors will be $O(P)$.*

Let $q' = 1 - (1 - q)^\alpha$ denote the probability that a given processor fails in α periods. As always, β is a constant from the interval $(0, 1)$. In case 2, once $1 - (1 + \beta)q' \geq \lambda$, at least $\frac{U}{\log U}$ processors remain operational for $\alpha \log U$ periods for $\alpha > 1$ with high probability (by Lemma 8). On the average, these processors that remain operational after α periods would have started at roughly equidistant positions (proved using Chernoff bounds). From this, we conclude that these processors will complete the pointer doubling of the whole array (except possibly for an initial segment), in time $O(\log U)$ with very high probability during Phase 1 of the PD-algorithm. The (possibly) unmarked initial segment will be processed during the second phase of PD-algorithm. Its length provides an upper bound on the time of the second phase.

Lemma 9 *The length of the unmarked initial segment of the array at the beginning of Phase 2 of the PD-algorithm is $O(\frac{U}{P} \log U)$ with probability at least $1 - U^{-b}$ for some $b > 1$.*

Theorem 3 *For $U/\log U \leq P \leq U$, the parallel time of the PD-algorithm is $O(\frac{U}{P} \log U)$ with probability at least $1 - U^{-b}$ for some $b > 1$, and the expected parallel time is $O(\frac{U}{P} \log U)$.* $\in$

7 Randomization can decrease the work to $O(U \log \log U)$

We will briefly sketch the CCC-algorithm in section 7.1. In doing this, our main purpose is to use it in outlining the ACC-algorithm (in section 7.2) that has surprisingly low overhead. It solves the Write-All problem in expected work $O(U \log \log U)$ with as many as $P \leq U/\log U$ processors.

7.1 The CCC-algorithm

The CCC-algorithm is a trivial variant of the maximum finding algorithm in asynchronous PRAMs, presented by Martel et al. in [MPS89]. Informally, view the locations in the array x of the Write-All problem as the U leaves of a binary tree, that is $\log U$ deep. (W.l.o.g. we assume that U is a power of two). The CCC-algorithm proceeds as follows. Initially all tree nodes are *unmarked*. We start with $P = U$ processors. Each functional processor selects a tree node at random. If the node v is a leaf or if the children of the node are marked, then node v is also

marked. This step is repeated by all the functional processors until the root is marked. Note that marking the root is the same as certifying that all the locations of x have been written. A simple variant of the analysis in [MPS89] shows that this algorithm does $O(U \log U)$ expected work. We can show that it has an interesting threshold in its parallel time behavior. Because of space constraints, we are unable to go into details of this issue here. Instead, we move on to analyze the ACC-algorithm, that has better expected work behavior, than any of the algorithms analyzed thus far.

7.2 The ACC-algorithm

Once again, as in Martel et al. [MPS89], we start with $U/\log U$ processors and divide the array $x[1..U]$ to be marked into $U/\log U$ subarrays each of size $\log U$. Each $\log U$ -sized subarray is now treated as the leaf of a full binary tree of $2U/\log U - 1$ nodes. The ACC-algorithm simply involves running the CCC-algorithm on the new tree, where now marking a leaf of the tree implies setting $x[i] := 1$ for all positions of the corresponding subarray.

Surprisingly, this simple modification to the CCC-algorithm only does an expected work of $O(U \log \log U)$ with as many as $U/\log U$ processors. Martel et al. also attempt an analysis of this algorithm in [MPS89] but their analysis appears to be flawed.

Theorem 4 *The expected work done by the ACC-algorithm to solve the Write-All problem with $P \leq U/\log U$ processors is $O(U \log \log U)$.*

8 A lower bound on the work of any deterministic algorithm for the Write-All problem

Theorem 5 *Given any deterministic P -processor CRCW PRAM algorithm that solves the Write-All problem, an adversary can force Byzantine fail-stop errors that result in $\Omega(U + P \log U)$ work being done by the algorithm.*

9 Acknowledgements

We are grateful to Samir Khuller for pointing out an omission in the first version of the TIES, for bringing reference [MPS89] to our attention, and for several helpful discussions. We are also grateful to Paris Kanellakis for bringing the issue of robustness in synchronous parallel computing to our attention.

References

- [A88] B. Awerbuch, "On the effects of feedback in dynamic network protocols," *Proc. of the 29th IEEE FOCS*, pp. 231-242, 1988.
- [AAG87] Y. Afek, B. Awerbuch, and E. Gafni, "Applying static network protocols to dynamic networks," *Proc. of the 28th IEEE FOCS*, pp. 358-370, 1987.
- [AAPS87] Y. Afek, B. Awerbuch, S. Plotkin, and M. Saks, "Local management of a global resource in a communication network," *Proc. of the 29th IEEE FOCS*, pp. 347-357, 1987.
- [AHU74] A. Aho, J. Hopcroft, and J. Ullman, "The design and analysis of computer algorithms," Addison-Wesley, 1974.
- [AS88] B. Awerbuch and M. Sipser, "Dynamic networks are as fast as static networks," *Proc. of the 29th IEEE FOCS*, pp. 206-219, 1988.
- [C52] H. Chernoff, "A measure of asymptotic efficiency for test of a hypothesis based on the sum of observations," *Annals of Math. Stat.*, vol. 23, pp. 493-509, 1952.
- [C86] R. Cole, "Parallel merge sort," *Proc. 27th IEEE FOCS*, pp. 511-516, 1986.
- [CV86] R. Cole and U. Vishkin, "Approximate and exact parallel scheduling with application to list, tree, and graph problems," *Proc. 27th IEEE FOCS*, pp. 468-491, 1986.
- [CZ89] R. Cole and O. Zajicek, "The APRAM: incorporating asynchrony into the PRAM model," *Proc. 89'SPAA*, pp. 170-178, 1989.
- [DPPU86] C. Dwork, D. Peleg, N. Pippinger, and E. Upfal, "Fault tolerance in networks of bounded degree," *Proc. of the 18th ACM STOC*, pp. 370-379, 1986.
- [FW78] S. Fortune and J. Wyllie, "Parallelism in random access machines," *Proc. 10th ACM STOC*, pp. 114-118, 1978.
- [Kh89] S. Khuller, private communication, 1989.
- [KLP89] Z. Kedem, G. Landau, and K. Palem, "Optimal parallel suffix-prefix matching algorithm and applications," *Proc. 89'SPAA*, 388-391, 1989.
- [KP88] Z. Kedem and K. Palem, "Optimal parallel algorithms for forest and term matching," *to appear in Theoretical Computer Science*.
- [KS89] P. Kanellakis and A. Shvartsman, "Efficient parallel algorithms can be made robust," preprint, October 24, 1989. (Initial version appeared in *Proc. 8th ACM PODC*, pp. 211-222, 1989.)
- [MPS89] C. Martell, A. Park, and R. Subramonian, "Fast asynchronous algorithms for shared memory parallel computers," *Tech. Rep. CSE-89-8, Univ. of California - Davis*, pp. 1-17, 1989.
- [P85] N. Pippinger, "On networks of noisy gates," *Proc. of the 26th IEEE FOCS*, pp. 30-38, 1985.
- [SS83] R. Schlichting and F. Schneider, "Fail-stop processors: an approach to designing fault-tolerant computing systems," *ACM Trans. Comput. syst.*, vol. 1, no. 3, pp. 222-238, 1983.
- [Shv89] A. Shvartsman, "Fault tolerant parallel computation," forthcoming Ph.D. thesis, Brown University.

- [TV84] R. Tarjan and U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time," *Proc. of the 25th IEEE FOCS*, pp. 12-22, 1984.

c.1

c.1

This book may be kept

A fine will be charged for each day the book is kept overtime.

GAYLORO 142

PRINTED IN U S A

